Böttinger *EURASIP Journal on Information Security* (2016) 2016:28
DOI 10.1186/s13635-016-0052-1

EURASIP Journal on Information Security

RESEARCH Open Access

Fuzzing binaries with Lévy flight swarms

Konstantin Böttinger

Abstract

We present a new method for random testing of binary executables inspired by biology. In our approach, we introduce the first fuzzer based on a mathematical model for optimal foraging. To minimize search time for possible vulnerabilities, we generate test cases with Lévy flights in the input space. In order to dynamically adapt test generation behavior to actual path exploration performance, we define a suitable measure for quality evaluation of test cases. This measure takes into account previously discovered code regions and allows us to construct a feedback mechanism. By controlling diffusivity of the test case generating Lévy processes with evaluation feedback from dynamic instrumentation, we are able to define a fully self-adaptive fuzzing algorithm. We aggregate multiple instances of such Lévy flights to fuzzing swarms which reveal flexible, robust, decentralized, and self-organized behavior.

Keywords: Self-adaptive random testing, Fuzzing, Lévy flights

1 Introduction

As software ever increases in size and complexity, we face the significant challenge to validate the systems surrounding us. Penetration testing of software has come a long way from its origins and nowadays shows an extensive diversity of possible strategies. All of them have the common aim to achieve maximal code coverage by generating suitable program inputs, also called test cases. Possible approaches range from dynamic symbolic [1, 2] and concolic [3–5] execution to more or less random testing using generational, mutational, black-box, or white-box fuzzers [6, 7]. Within the latter domain of random test generation, current strategies for input generation basically rely on heuristics and sophisticated guessing. It is still an open question how to optimally generate inputs that trigger a maximum number of bugs in a finite amount of time.

In the course of researching new effective search strategies, we find similar problems in biology, particularly in the field of optimal foraging. A variety of biological systems let us observe optimal strategies for finding energy sources by simultaneously avoiding predators. When we identify sources of food with possible vulnerabilities in binary executables and predators with the overhead of execution runtime, we are inspired to adapt mathematical models of optimal foraging to test case generation. This approach enables us to take stochastic models of optimal foraging as a basis for input mutation. In particular, we rely on Lévy flights to search for bug triggering test cases in input space.

Before summarizing our contributions, we first give some short background on fuzzing, optimal foraging, and the Lévy flight hypothesis.

1.1 Fuzzing

There exists a substantial diversity of test case generation strategies for random testing binaries. All these approaches have in common to a greater or lesser extent the random generation of test cases with the aim of driving the targeted program to an unexpected and possibly exploitable state. The most significant advantage of fuzzing is its ease of use. Most executable binaries that process any input data are suitable targets for random test generation, and effective fuzzers are implemented in a short time.

1.2 Optimal foraging

Observing biological systems has led to speculation that there might be simple laws of motion for animals searching for sources of energy in the face of predators. Regardless of whether we look at bumblebees [8], fish and hunting marine predators in the sea [9, 10], gray seals [11], spider monkeys [12], the flight search patterns of albatrosses [13], the wandering of reindeer [14], the reaction pathways of DNA-binding proteins [15], or the neutralization of pathogens by white blood cells [16], we can

Correspondence: konstantin.boettinger@aisec.fraunhofer.de
Fraunhofer Institute for Applied and Integrated Security, Parkring 4, 85748 Garching, Germany

discover emerging movement patterns all those examples have in common. Mathematically modeling such common patterns is an active field of research in biology and is more generally referred to as *movement ecology*. While the physics of foraging [17] provides us several possible models, our choice is not guided by accuracy with respect to the biological process but by minimization of software bug search time. This leads us to the special class of stochastic processes called *Lévy flights* which we discuss in more detail in Section 3.

1.3 Lévy flight hypothesis

Within the variety of models for optimal foraging, Lévy flights have several characteristic properties that show promise for software testing. In particular, the Lévy flight hypothesis accentuates the most significant property of these kinds of stochastic processes for our purposes. It states that Lévy flights minimize search time when foraging sources of food that are sparsely and randomly distributed, resting, and refillable. These assumptions match the properties of bugs in software (with the interpretation that *refillable* translates to the fact that software bugs stay until fixed). In addition to the mathematical Lévy flight hypothesis, the Lévy flight *foraging* hypothesis in theoretical biology states that these processes actually model real foraging behavior in certain biological systems due to natural selection. The Lévy flight hypothesis constitutes the major connection link between optimal foraging theory and random software testing.

1.4 Swarm behavior

While moving patterns of foraging animals inspire us to define the behavior of a single fuzzer, we are further guided by biology when accumulating multiple fuzzer instances to a parallelized testing framework. Again, we take a look at nature to discover a whole branch of science that researches swarm behavior [18]. For example, the ants of a colony collectively find the shortest path to a food source. Based on simple rules for modeling natural swarm behavior, we construct a *fuzzing swarm* that mimics colony clustering observed in biology. Our algorithm navigates the fuzzing swarm without a central control and provides self-organization of the fuzzers as they flexibly adapt to the binary structure under test.

In this paper, we propose a novel method for random software testing based on the theory of optimal foraging. In summary, we make the following contributions:

- We introduce a novel fuzzing method based on Lévy flights in the input space in order to maximize coverage of execution paths.
- We define a suitable measure for quality evaluation of test cases in input space with respect to previously explored code regions.
- In order to control diffusivity of the test generation processes, we define a feedback mechanism connecting current path exploration performance to the test generation module.
- We enable self-adaptive fuzzing behavior by adjusting the Lévy flight parameters according to feedback from dynamic instrumentation of the target executable.
- We aggregate multiple instances of such Lévy flights to fuzzing swarms which reveal flexible, robust, decentralized, and self-organized behavior.
- We implement the presented algorithm to show the feasibility of our approach.

The remainder of this paper is organized as follows. In Section 2, we discuss related work. In Section 3, we present necessary background on Lévy flights and show how to construct them in input space. We define a quality measure for generated test cases in Section 4, introduce our self-adapting algorithm for individual fuzzers in Section 5, and construct a swarm of multiple fuzzing instances in Section 6. Next, we give details regarding our implementation in Section 7 and discuss properties, possible modifications, and expansions of the proposed algorithm in Section 8. The paper concludes with a short outlook in Section 9.

2 Related work

This paper is an extension of *Hunting Bugs with Lévy Flight Foraging* [19]. The prevalent method used for binary vulnerability detection is random test generation, also called fuzzing. Here, inputs are randomly generated and injected into the target program with the aim to gain maximal code coverage in the execution graph and drive the program to an unexpected and exploitable state. There is a rich diversity of fuzzing tools available, each focusing on specialized approaches. Multiple taxonomies for random test generation techniques have been proposed, and the most common is classification into mutational or generational fuzzing. Mutation fuzzers are unaware of the input format and mutate the whole range of input variables blindly. In contrast, generation fuzzers take the input format into account and generate inputs according to the format definition. For example, generation fuzzers can be aware of the file formats accepted by a program under test or the network protocol definition processed by a network stack implementation. We can further classify random test generation methods into black-box or white-box fuzzing, depending on the awareness of execution traces of generated inputs. We refer to [6, 7] for a comprehensive account.

For definition of our quality measure for test cases, we built upon executable code coverage strategies. The idea to generate program inputs that maximize execution path

coverage in order to trigger vulnerabilities has been discussed in the field of test case prioritization some time ago, see e.g., [20, 21] for a comparison of coverage-based techniques. Rebert et al. [22] discuss and compare methods to gain optimal seed selection with respect to fuzzing, and their findings support our decision to select code coverage for evaluating the quality of test cases. The work of Cha et al. [23] is distantly related to a substep of our approach in the sense that they apply dynamic instrumentation to initially set the mutation ratio. However, they use completely different methods based on symbolic execution. Since symbolic preprocessing is very cost-intensive, they further compute the mutation ratio only once per test.

Lévy flights have been studied extensively in mathematics, and we refer to Zaburdaev et al. [24] and the references therein for a comprehensive introduction to this field. Very recently, Chupeau et al. [25] connected Lévy flights to optimal search strategies and minimization of cover times.

3 Lévy flights in input space

In this section, we give the necessary background on Lévy flights and motivate their application. With this background, we then define Lévy flights in input space.

3.1 Lévy flights

Lévy flights are basically random walks in which step lengths exhibit power law tails. We aim for a short and illustrative presentation of the topic and refer to Zaburdaev et al. [24] for a comprehensive introduction. Pictorially if a particle moves stepwise in space while randomly choosing an arbitrary new direction after each step, it describes a Brownian motion. If in addition the step lengths of this particle vary after each step and are distributed according to a certain power law, it describes a Lévy flight.

Formally, Lévy processes comprise a special class of Markovian stochastic processes, i.e., collections of random variables

$$(L_t),\ t \in T \tag{1}$$

defined on a sample space Ω of a probability space $(\Omega, \mathcal{F}, P)$, mapping into a measurable space $(\Omega', \mathcal{F}')$, and indexed by a totally ordered set T. In our case, Ω' refers to the discrete input space of the program and the index *time* T models the discrete iterations of test case generation, so we can assume $T = \mathbb{N}$. The process $(L_t)_{t \in T}$ is said to have *independent increments* if the differences

$$L_{t_2} - L_{t_1}, L_{t_3} - L_{t_2}, ..., L_{t_n} - L_{t_{n-1}} \tag{2}$$

are independent for all choices of $t_1 < t_2 < ... < t_n \in T$. The process $(L_t),\ t \in T$ is said to be *stationary*, if

$$\forall t_1, t_2 \in T, h > 0 :\ L_{t_1+h} - L_{t_1} \sim L_{t_2+h} - L_{t_2}, \tag{3}$$

i.e., increments for equal time intervals are equally distributed. A Lévy process is then formally defined to be a stochastic process having independent and stationary increments. The additional property

$$L_0 = 0\ a.s. \tag{4}$$

(i.e., almost surely) is sometimes included in the definition, but our proposed algorithm includes starting points other than the origin.

To construct a Lévy process, $(L_n)_{n \in \mathbb{N}}$ we simply sum up independent and identically distributed random variables $(Z_n)_{n \in \mathbb{N}}$, i.e.,

$$L_n := \sum_{i=1}^{n} Z_i. \tag{5}$$

The process $(L_n)_{n \in \mathbb{N}}$ is Markovian in the sense that

$$P(L_n = x_n | L_{n-1} = x_{n-1}, ..., L_0 = x_o) \tag{6}$$

$$= P(L_n = x_n | L_{n-1} = x_{n-1}), \tag{7}$$

which simplifies a practical implementation. If the distribution of step lengths in a Lévy process is heavy-tailed, i.e., if the probability is not exponentially bounded, we call the process a *Lévy flight*. Such processes generalize Brownian motion in that their flight lengths l are distributed according to the power law

$$p(l) \sim |l|^{-1-\alpha}, \tag{8}$$

where $0 < \alpha < 2$. They exhibit infinite variance

$$< l^2 >= \infty \tag{9}$$

which practically results in sometimes large jumps during search process. In fact, the ability to drive a particle very long distances within a single step gives Lévy flights their name. While Brownian motion is a suitable search strategy for densely distributed targets, Lévy flights are more efficient than Brownian motion in detecting widely scattered (software) bugs. Although there is much to say about the theoretical aspects of this class of stochastic processes, we basically refer to the power law in Eq. (8) in the following. Smaller values of α yield a heavier tail (resulting in frequent long flights and super-diffusion), whereas higher values of α reveal a distribution with probability mass around zero (resulting in frequent small steps and sub-diffusion). In Section 5, we adapt α according to feedback information from dynamic instrumentation of the targeted binary.

As indicated in Section 1, Lévy flights are directly connected to the minimal time it takes to cover a given search domain. We refer to [25] for recent results regarding minimization of the mean search time for single targets.

3.2 Input space flights

Next, we construct Lévy flights in the input space of binary executables under test. Therefore, assume the input to be a bit string of length N. If we simply wanted an optimal search through the input space without any boundary conditions, we would construct a one-dimensional Lévy flight in the linear space $\{0, ..., 2^N\}$. However, our aim is not input space coverage but execution code coverage of the binary under test. In this section, we construct a stochastic process in input space with the properties we need for the main fuzzing algorithm presented in Section 5.

First, we divide the input into n segments of size $m = \frac{N}{n}$ (assuming without loss of generality that N is a multiple of n). We then define two Lévy processes, one in the space of offsets $\mathcal{O} = \{1, ..., n\}$ and one in the space of segment values $\mathcal{S} = \{1, ..., 2^m\}$. With underlying probability spaces $(\Omega_1, \mathcal{F}_1, P_1)$ and $(\Omega_2, \mathcal{F}_2, P_2)$, we define the one-dimensional Lévy flights

$$L_t^1 : \Omega_1 \to \mathcal{O} \tag{10}$$

$$L_t^2 : \Omega_2 \to \mathcal{S} \tag{11}$$

with index space $t \in \mathbb{N}$ and corresponding power law distribution of flight lengths l

$$p_j(l) \sim |l|^{-1-\alpha_i},\ j = 1, 2, \tag{12}$$

where $0 < \alpha_i < 2$. While $(L_t^1)_{t\in\mathbb{N}}$ performs a Lévy flight in the offset parameter space, $(L_t^2)_{t\in\mathbb{N}}$ performs Lévy flights within the segment space indicated by the offset. Regarding the initial starting point (L_0^1, L_0^2), we assume a given seed input. We choose an arbitrary initial offset $L_0^1 \in \mathcal{O}$ and set the initial value of L_0^2 according to the segment value (with offset L_0^1) of the seed input.

By setting different values of α, we can control the diffusivity of the stochastic processes $(L_t^1)_{t\in\mathbb{N}}$ and $(L_t^2)_{t\in\mathbb{N}}$. If we find a combination of offset and segment values of high quality, the fuzzer should automatically explore nearby test cases, which is realized by higher values of $0 < \alpha_i < 2$. Similarly, if the currently explored region within input space reveals low quality test cases, the fuzzer should automatically adapt to widen its search pattern by decreasing α. Therefore, we first have to define a quality measure for test cases.

4 Quality evaluation of test cases

In this section, we define a quality measure for generated test cases. We aim for maximal possible code coverage in a finite amount of time, so we evaluate a single input by its ability to reach previously undiscovered execution paths. In other words, if we generate an input that drives the program under test to a new execution path, this input gets a high-quality rating. Therefore, we have to define a similarity measure for execution traces. We will then use this measure in Section 5 as feedback to dynamically adapt diffusivity of the test case generation process.

The field of test case prioritization provides effective methods for coverage-based rating (see [20, 21] for a comparison). We adapt the method of prioritizing test cases by additional basic block coverage. As introduced in Section 3, we assume inputs for the program under test to be bit strings of size N and denote the space of all possible inputs as $\mathcal{I} = \{0, ..., 2^N\}$. Our challenge can then be formulated as follows. Given a subset of already generated input values $\mathcal{I}' \subset \mathcal{I}$, how do we measure the quality of a new input $x_0 \in \mathcal{I}$ with respect to maximal code coverage? For a given $x_0 \in \mathcal{I}$, let c_{x_0} denote the execution path the program takes for processing x_0. Intuitively, we would assign a high-quality rating to the new input x_0 if it drives the targeted program to a previously undiscovered execution path, i.e., if c_{x_0} differs significantly from all previously explored execution paths $\{c_x | x \in \mathcal{I}'\}$. To measure this path difference, we take the amount of newly discovered basic blocks into account. Here, we refer to a *basic block* as a sequence of machine instructions without branch instructions between block entry and block exit. Let $B(c_x)$ denote the set of basic blocks of execution path c_x. The set of newly discovered basic blocks while processing a new test case x_0 given already executed test cases $\mathcal{I}' \subset \mathcal{I}$ is then

$$B(c_{x_0}) \setminus \left(\bigcup_{x\in\mathcal{I}'} B(c_x) \right). \tag{13}$$

We define the number $E(x_0, \mathcal{I}')$ of these newly discovered blocks as

$$E(x_0, \mathcal{I}') := \left| B(c_{x_0}) \setminus \left(\bigcup_{x\in\mathcal{I}'} B(c_x) \right) \right|, \tag{14}$$

where $|A|$ denotes the number of elements within a set A. The number $E(x_0, \mathcal{I}')$ indicates the number of newly discovered basic blocks when processing x_0 with respect to the already known basic blocks executed by the test cases within $\mathcal{I}'$. Intuitively, $E(x_0, \mathcal{I}')$ gives us a quality measure for input x_0 in terms of maximization of basic block coverage. In order to construct a feedback mechanism, we will use a slightly generalized version of this measure to control diffusivity of the input generating Lévy processes in our fuzzing algorithm in Section 5.

5 Fuzzing algorithm

In this section, we present the overall fuzzing algorithm. Our approach uses stochastic processes (i.e., Lévy flights as introduced in Section 3) in the input space to generate test cases. To steer the diffusivity of test case generation, we provide feedback regarding the quality of test cases (as defined in Section 4) to the test generation process in order to yield self-adaptive fuzzing.

We first prepend an example regarding the interplay between input space coverage and execution path coverage to motivate our fuzzing algorithm. Consider a program which processes inputs from an input space $\mathcal{I}$. Our aim is to generate a subset $\mathcal{I}' \subset \mathcal{I}$ of test cases (in finite amount of time) that yields maximal possible execution path coverage when processed by the target program. Further assume the program to reveal deep execution paths (covering long sequences of basic blocks) only for 3% of the inputs $\mathcal{I}$, i.e., 97% of inputs are inappropriate test cases for fuzzing. Since we initially cannot predict which of the test cases reveals high quality (determined by e.g., the execution path length or the number of different executed basic blocks), one strategy to reach good code coverage would be black-box fuzzing, i.e., randomly generating test cases within $\mathcal{I}$ hoping that we eventually hit some of the 3% high quality inputs. We could realize such an optimal search through input space with highly diffusive stochastic processes, i.e., Lévy flights as presented in Section 3.

As mentioned above, the Lévy flight hypotheses predicts an effective optimal search through input space due to their diffusivity properties. On the one hand, this diffusivity guarantees us reaching the 3% with very high probability. On the other hand, once we have reached input regions within the 3% of high quality test cases, the same diffusivity also guarantees us that we will leave them very efficiently. This is why we need to adapt the diffusivity of the stochastic process according to the quality of the currently generated test cases. If the currently generated test cases reveal high path coverage, the Lévy flight should be localized in the sense that it reduces its diffusivity to explore nearby inputs. In turn, if the currently generated test cases reveal only little coverage, diffusivity should increase in order to widen the search for more suitable input regions. By instrumenting the binary under test and applying the quality evaluation of test cases introduced in Section 4, we are able to feedback coverage information of currently explored input regions to the test case generation algorithm. In the following, we construct a self-adaptive fuzzing strategy that automatically expands its search when reaching low-quality input regions and focuses exploration when having the feedback of good code coverage.

5.1 Initial seed

We start with an initial non-empty set of input seeds $X_0 \subset \mathcal{I}$. As described in Section 3, we assume the elements $x \in X_0$ to be bit strings of length N and divide each of them into n segments of size $m = \frac{N}{n}$ (assuming without loss of generality that N is a multiple of n). Practically, the input seeds X_0 can be arbitrary files provided manually by the tester; they may not even be valid with regard to the input format of the program under test. We further set two initial diffusive parameters $0 < \alpha_1, \alpha_2 < 2$ and an initial offset $q_0 \in \{1, \ldots, n\}$.

5.2 Test case generation

The test case generation step takes as input a test case x_0, diffusion parameters α_1 and α_2, an offset number $q_0 \in \{1, \ldots, n\}$, and a natural number $k_{\text{gen}} \in \mathbb{N}$ of maximal test cases to be generated. It outputs a set X_{gen} of k_{gen} new test cases $X_{\text{gen}} \in \mathcal{I}$.

As introduced in Section 3, we refer to the offset space as $\mathcal{O} = \{1, \ldots, n\}$ and to the segment space as $\mathcal{S} = \{1, \ldots, 2^m\}$. We denote with $x_0(q_0)$ the segment value of input x_0 at offset q_0. For the Lévy flights

$$L_t^1 : \Omega_1 \to \mathcal{O} \tag{15}$$

in the offsets $\mathcal{O}$ and

$$L_t^2 : \Omega_2 \to \mathcal{S} \tag{16}$$

in $\mathcal{S}$ with flight lengths l distributed according to the power law

$$p_j(l) \sim |l|^{-1-\alpha_j},\ j = 1, 2, \tag{17}$$

we set the initial conditions

$$L_0^1 = q_0 \quad \text{and} \tag{18}$$

$$L_0^2 = x_0(q_0), \tag{19}$$

respectively. Let $R(x_0, q_0, s_0)$ denote the bit string generated by replacing the value $x_0(q_0)$ of bit string x_0 at offset q_0 by a new value s_0. Both stochastic processes $(L_t^1)_{t\in\mathbb{N}}$ and $(L_t^2)_{t\in\mathbb{N}}$ are then simulated for k_{gen} steps to generate the k_{gen} new test cases

$$x_1 := R\left(x_0, L_0^1, L_1^2\right) \tag{20}$$

$$x_2 := R\left(x_1, L_1^1, L_2^2\right) \quad \ldots \tag{21}$$

$$x_{t+1} := R\left(x_t, L_t^1, L_{t+1}^2\right) \quad \ldots \tag{22}$$

$$x_{k_{\text{gen}}} := R\left(x_{k_{\text{gen}}-1}, L_{k_{\text{gen}}-1}^1, L_{k_{\text{gen}}}^2\right). \tag{23}$$

For simplicity of notation in this definition, we identify the values L_t^j with their respective binary representations (as bit string). In words, we start with the initial test case x_0 and replace its segment content at offset $L_0^1 = q_0$ with the new value L_1^2, which is the value in segment space $\mathcal{S} = \{1, \ldots, 2^m\}$ that we get when taking a first random step with the Lévy flight $(L_t^2)_{t\in\mathbb{N}}$. This yields x_1. We get the next test case x_2 by considering the just generated x_1,

setting the offset according to $(L_t^2)_{t\in\mathbb{N}}$, and then replacing the content of the segment indicated by this offset by a new segment value chosen by $(L_t^2)_{t\in\mathbb{N}}$. We proceed with this algorithm until the set

$$X_{\text{gen}} := \{x_1, \ldots, x_{k_{\text{gen}}}\} \tag{24}$$

of k_{gen} new test cases is generated.

5.3 Quality evaluation

The quality evaluation step takes as input two sets of test cases $X_{\text{gen}}, \mathcal{I}' \subset \mathcal{I}$ and outputs a quality rating $\tilde{E}(X_{\text{gen}}, \mathcal{I}')$ of X_{gen} with respect to $\mathcal{I}'$. We already defined the number $E(x_0, \mathcal{I}')$ of newly discovered basic blocks for a single test case x_0 with respect to a given subset $\mathcal{I}' \subset \mathcal{I}$ in Eq. (14). To generalize this definition to a quality rating $\tilde{E}(X_{\text{gen}}, \mathcal{I}')$ of a set of test cases X_{gen} (with respect to $\mathcal{I}'$), we define the mean

$$\tilde{E}(X_{\text{gen}}, \mathcal{I}') := |X_{\text{gen}}|^{-1} \sum_{x\in X_{\text{gen}}} E(x, \mathcal{I}'). \tag{25}$$

5.4 Adaptation of diffusivity

The diffusivity adaptation step takes as input a quality rating $\tilde{E}(X_{\text{gen}}, \mathcal{I}') \in \mathbb{N}$, two parameters $b_1, b_2 \in \mathbb{R}^+$ (controlling the switching behavior from sub-diffusion to super-diffusion) and outputs two adapted parameters $0 < \alpha_1, \alpha_2 < 2$, which according to the power law (17) regulate the diffusivity of the Lévy flights $(L_t^1)_{t\in\mathbb{N}}$ and $(L_t^2)_{t\in\mathbb{N}}$.

Our aim (as motivated at the beginning of this section) is to adapt the diffusion parameters in such a way that the algorithm automatically focuses its search (by decreasing diffusivity of the generating Lévy flights) when generating high-quality (i.e., high coverage) test cases and in turn automatically widens its search (by increasing diffusivity) in the case of low-quality (i.e., low coverage) test cases. As discussed in Section 3, we can control diffusivity by setting suitable values of α_1 and α_2. Smaller diffusivity parameters result in frequent long flights and super-diffusion, whereas higher parameters reveal frequent small steps and sub-diffusion. To achieve this, we select a monotonically increasing function $f : \mathbb{R} \to (0, 2)$ with $f(0) \leq \epsilon$ (for $\epsilon > 0$ sufficiently small) and $\lim_{t\to\infty} f(t) = 2$. Any such function will provide self adaptation of diffusivity of the Lévy flights, and we simply choose two functions

$$f_i(t) := \frac{2}{1 + e^{b_i - t}},\ i = 1, 2, \tag{26}$$

where $b_i \in \mathbb{R}^+$ are fixed parameters that determine at which point within the quality rating spectrum (i.e., at which mean number of newly discovered basic blocks) the search behavior of $(L_t^1)_{t\in\mathbb{N}}$ and $(L_t^2)_{t\in\mathbb{N}}$ switches from sub-diffusion to super-diffusion. With this function, we adapt diffusivity to

$$\alpha_i = f\left(\tilde{E}(X_{\text{gen}}, \mathcal{I}')\right),\ i = 1, 2. \tag{27}$$

The next iteration of test case generation is then executed with adapted Lévy flights.

5.5 Test case update

This step takes as input two sets of test cases $X_{\text{old}}, X_{\text{gen}} \subset \mathcal{I}$ and outputs an updated set of test cases X_{new}. During the fuzzing process, we generate a steady stream of new test cases which we directly evaluate with respect to the set of previously generated inputs (as discussed in the quality evaluation step). However, if we archive every single test case and for each generation step evaluate the k_{gen} currently generated new test cases against the whole history of previously generated test cases, fuzzing speed decays constantly with increasing duration of the fuzzing campaign. Therefore, we define an upper bound $k_{\max} \in \mathbb{N}$ of total test cases that we keep for quality evaluation of new test cases. Small values of $k_{\max}$ may cause the Lévy flights $(L_t^1)_{t\in\mathbb{N}}$ and $(L_t^2)_{t\in\mathbb{N}}$ to revisit already explored input regions without being adapted (by decreasing the parameters α_i) to perform super-diffusion and widen their search behavior. However, this causes no problem due to the Lévy flight hypothesis (discussed in Section 1).

The update of X_{old} with X_{gen} simply follows a *first in, first out* strategy. Initially if $|X_{\text{old}}| + |X_{\text{new}}| < k_{\max}$, we append all newly generated test cases so that $X_{\text{new}} = X_{\text{old}} \cup X_{\text{gen}}$. Otherwise, we first delete the oldest k_{old} entries in X_{old}, where

$$k_{\text{old}} = |X_{\text{old}}| + |X_{\text{new}}| - k_{\max}, \tag{28}$$

and then take the union.

5.6 Joining the pieces

Now that we have presented all individual parts, we can combine them. The overall fuzzing algorithm is depicted in Fig. 1.

The initial seed generation step outputs a non-empty set of test cases $X_0 \subset \mathcal{I}$, two diffusivity parameters α_1 and α_2, and an initial offset q_0. The inputs X_0 are added to the list of test cases X_{all}. Then, the fuzzer enters the loop of test case generation, quality evaluation, adaptation of diffusivity, and test case update. The first step within the loop (referred to as $\text{Last}(X_{\text{all}})$) sets q_0 to the last reached offset position of $(L_t^1)_{t\in\mathbb{N}}$. In the first invocation of $\text{Last}(X_{\text{all}})$), this is simply the already given seed offset, in all subsequent invocations q_0 is updated to the last state of $(L_t^1)_{t\in\mathbb{N}}$. The Last() function also selects the most recently added test case x_0 in X_{all}, which gives the initial condition for $(L_t^2)_{t\in\mathbb{N}}$ in the generation step. In our implementation, we realize the *Last*() function by retaining the reached states of both processes $(L_t^1)_{t\in\mathbb{N}}$ and $(L_t^2)_{t\in\mathbb{N}}$ between simulations.

Starting at $L_0^1 = q_0$ and $L_0^2 = x_0(q_0)$, the Lévy flights $(L_t^1)_{t\in\mathbb{N}}$ and $(L_t^2)_{t\in\mathbb{N}}$ generate the set of new inputs X_{gen} by diffusing through input space with diffusivity α_1 and α_2,

Input: Parameters $b_1,\ b_2,\ k_{gen},\ k_{max}$

$X_{all} = \emptyset$
$X_0,\ \alpha_1,\ \alpha_2,\ q_0 \leftarrow \text{Seed}()$
append X_0 to X_{all}
do:
$q_0, x_0 \leftarrow \text{Last}(X_{all})$
$X_{gen} \leftarrow \text{Gen}(x_0, \alpha_1, \alpha_2, q_0, k_{gen})$
$\tilde{E} \leftarrow \text{Eval}(X_{gen}, X_{all})$
$\alpha_1, \alpha_2 \leftarrow \text{Adapt}(\tilde{E}, b_1, b_2)$
$X_{all} \leftarrow \text{Update}(X_{gen}, X_{all}, k_{max})$
while (true)

Fig. 1 Individual fuzzing algorithm. After initial seed generation, the fuzzer enters the loop of test case generation, quality evaluation, adaptation of diffusivity, and test case update

respectively. The quality of X_{gen} is then evaluated against the previous test cases in X_{all}. Depending on the quality rating outcome, the diffusivity of $(L_t^1)_{t\in\mathbb{N}}$ and $(L_t^2)_{t\in\mathbb{N}}$ is then adapted correspondingly by updating α_1 and α_2 according to the sigmoid functions f_i in Eqs. (26). Then the current list of test cases X_{all} is updated with the just generated set X_{gen} and the fuzzer continues to loop.

Regarding complexity of the fuzzing algorithm we note that all of the individual parts are processed efficiently in the sense that their time complexity is bound by a constant. Especially the evaluation step *Eval()* is designed to scale: in the first iterations of the loop, the cost of evaluating X_{gen} against X_{all} is bound by $\mathcal{O}(|X_{\text{all}}|^2)$. To counter this growth, we defined an upper bound $k_{\max} \in \mathbb{N}$ for $|X_{\text{all}}|$ in the *test case update* step above.

6 Lévy flight swarms

Now that we have constructed an individual fuzzing process, we can aggregate multiple instances of such processes to *fuzzing swarms*. Each individual basically performs the search algorithm described in Section 5, but receives additional information from its neighbors and adapts accordingly. Adaptation rules are inspired by social insect colonies [18] and provide a flexible, robust, decentralized, and self-organized swarm behavior as described in Section 1.

With the probability spaces $(\Omega_i, \mathcal{F}_i, P_i)$ $(i = 1, 2)$ as defined in Section 3, let

$$\mathcal{F}_1 \otimes \mathcal{F}_2 = \sigma(\mathcal{F}_1 \times \mathcal{F}_2) \tag{29}$$

denote the σ-algebra generated by the Cartesian product $\mathcal{F}_1 \times \mathcal{F}_2$, i.e., the smallest σ-algebra which contains the sets in $\mathcal{F}_1 \times \mathcal{F}_2$. The flight of an individual fuzzer

$$(F_t)_{t\in\mathbb{N}} := \left(L_t^1, L_t^2\right)_{t\in\mathbb{N}} \tag{30}$$

is formally defined on the product space

$$(\Omega_1 \times \Omega_2, \mathcal{F}_1 \otimes \mathcal{F}_2, P_1 \times P_2), \tag{31}$$

where $P_1 \times P_2$ denotes the corresponding product measure. We can then define a swarm S of d individual flights

$$S := \left\{F^i, \mid i = 1, \ldots, d\right\}, \tag{32}$$

each of which performs the loop of test case generation, quality evaluation, adaptation of diffusivity, and test case update as described in Section 5. For each loop iteration, the individuals F^i of the swarm S generate test cases

$$X_{\text{gen}}^i := \left\{x_1^i, \ldots, x_{k_{\text{gen}}}^i\right\}, \tag{33}$$

and each individual maintains its own version of aggregated test cases X_{all}^i.

To perform collective fuzzing, there are several possibilities for the individuals F^i of the swarm S to exchange information. One strategy would be to define a shared set of already generated test cases $\bigcup_i X_{\text{all}}^i$ which could be seen as a global shared memory of already generated test cases. To keep the cost of each evaluation step *Eval()* low, we defined an upper bound $k_{\max} \in \mathbb{N}$ for $|X_{\text{all}}^i|$ in Section 5. If all swarm individuals add their generated test cases to the global shared memory, this would result in a high cost for each individual to evaluate their newly generated test cases X_{gen}^i against $\bigcup_i X_{\text{all}}^i$, since the complexity of *Eval()* is bound by $\mathcal{O}(|\bigcup_i X_{\text{all}}^i|^2)$.

Therefore, we explore another strategy to share information between swarm individuals. Intuitively, after a fixed amount of search time, each F^i of the swarm S receives the actual quality evaluation $\tilde{E}$ of its neighbors and jumps to the one neighbor which is currently searching the most promising input area. If an individual $F_\lambda \in S$ is searching an input area of highest quality $\tilde{E}_\lambda$ test cases among its nearby swarm individuals, all neighbors with lower values of $\tilde{E}_\lambda$ jump to the current position of F_λ in input space. We will formalize this idea in the following, where the index λ refers to local maxima of test case quality $\tilde{E}$.

We first need a metric in input space in order to consider neighborhoods of swarm individuals. As a natural metric in the space of all possible inputs $\mathcal{I} = \{0, \ldots, 2^N\}$, we choose the Hamming distance δ: two bit strings $x = (x_1, \ldots, x_N)$ and $x' = (x'_1, \ldots, x'_N)$ of size N then have distance

$$\delta(x, x') := \left|\left\{j \in 1, \ldots, N \mid x_j \neq x'_j\right\}\right|. \tag{34}$$

We can then simply measure the distance $\delta_S\left(F_t^i, F_t^j\right)$ of two individuals

$$F^i = \left(L^{1,i}, L^{2,i}\right) \in S \tag{35}$$

$$F^j = \left(L^{1,j}, L^{2,j}\right) \in S \tag{36}$$

at time $t \in \mathbb{N}$ with

$$\delta_S\left(F_t^i, F_t^j\right) := \delta\left(x_t^i, x_t^j\right), \tag{37}$$

where

$$x_t^i = R\left(x_{t-1}^i, L_{t-1}^{1,i}, L_t^{2,i}\right) \tag{38}$$

$$x_t^j = R\left(x_{t-1}^j, L_{t-1}^{1,i}, L_t^{2,i}\right) \tag{39}$$

are defined as in Eq. (22). In words, the distance $\delta_S\left(F_t^i, F_t^j\right)$ of two swarm individuals $F^i, F^j \in S$ at a certain time $t \in \mathbb{N}$ is the Hamming distance of the, respectively, two test cases generated at time t.

With this metric, we could proceed with considering the R-neighborhood

$$U_R(F_0) := \{F \in S \mid \delta_S(F_0, F) < R\} \tag{40}$$

of a swarm individual $F_0 \in S$ for an arbitrary $R \in \mathbb{N}$. However, this definition of neighborhood would result in high processing costs for large swarms: each individual must calculate the distances to all other individuals of the swarm before jumping to the position of the neighbor individual which generated test cases of highest quality $\tilde{E}_\lambda$. Therefore, we introduce a more lightweight method of calculating neighborhoods that scales to large swarms. We periodically divide the whole swarm S into k clusters using a k-means clustering algorithm to yield the disjoint partition $S = \dot{\bigcup}_k C_k$. Each individual $F^i \in C_j$ then only takes into account the test case quality $\tilde{E}$ of individuals within the same cluster C_j before relocation.

The overall swarm fuzzing algorithm is depicted in Fig. 2. The first part initializes the d swarm individuals F^i $(i = 1, ..., d)$. Each of the d initializations in the first *for* loop basically corresponds to the single fuzzer setup described in Section 5, with the minor formal difference that the *Init*() function randomly selects d inputs $x_0^i \in X_0^i \subset I$ $(i = 1, ..., d)$ among the seed input sets to fix the starting points of the F^i.

The algorithm then enters the main *do-while* loop, which consists of three parts: fuzzing, clustering, and relocation. First, all F^i $(i = 1, ..., d)$ start fuzzing the binary performing test case generation, quality evaluation, adaptation of diffusivity, and test case update as described in Section 5.

Second, the *Cluster*() function divides the swarm S into k clusters C_j $(j = 1, \ldots, k)$ as described above. We refer to a single cluster as the *neighborhood* of the swarm individuals belonging to this cluster. Swarm individuals F^i mutating on nearby inputs (measured with the Hamming metric) are assigned to the same cluster, whereas distant populations share different neighborhoods.

Input: Parameters $d,\ k,\ b_1,\ b_2,\ k_{gen},\ k_{max}$

for $i = 1, ..., d$ **:**
- $X_{all}^i = \emptyset$
- $X_0^i,\ \alpha_1^i,\ \alpha_2^i,\ q_0^i \leftarrow$ Seed()
- append X_0^i to X_{all}^i
- $x_0^i \leftarrow \mathrm{Init}(X_0^i)$

do:
- **for** $i = 1, ..., d$ **:**
  - $X_{gen}^i \leftarrow \mathrm{Gen}(x_0^i, \alpha_1^i, \alpha_2^i, q_0^i, k_{gen})$
  - $\tilde{E}^i \leftarrow \mathrm{Eval}(X_{gen}^i, X_{all}^i)$
  - $\alpha_1^i, \alpha_2^i \leftarrow \mathrm{Adapt}(\tilde{E}^i, b_1, b_2)$
  - $X_{all}^i \leftarrow \mathrm{Update}(X_{gen}^i, X_{all}^i, k_{max})$
- $C_1, ..., C_k \leftarrow \mathrm{Cluster}(x_{k_{gen}}^1 \in X_{gen}^1, ..., x_{k_{gen}}^d \in X_{gen}^d)$
- $x_0^1, q_0^1, ..., x_0^d, q_0^d \leftarrow \mathrm{Relocate}(C_1, ..., C_k)$
- **while** (true)

Fig. 2 Swarm fuzzing algorithm. The swarm of fuzzers enters the loop of individual fuzzing, clustering with k-means, and relocation of individuals to positions of highest test case quality within respective clusters

Third, all swarm individuals F^i within the same neighborhood C_j are relocated to the most promising nearby search position. For each cluster C_j $(j = 1, \ldots, k)$ the *Relocate*() function compares the current test case quality $\tilde{E}^i$ of all F^i within the same neighborhood. Without loss of generality there is one swarm individual $F_\lambda^j \in C_j$ in each neighborhood C_j with maximal quality evaluation $\tilde{E}_\lambda^j$ (in the case of multiple neighbors having the same $\tilde{E}$ we simply could choose one of them randomly). Then, the *Relocate*() function resets the initial positions of all $F^i \in C_j$ to

$$L_0^{1,i} \leftarrow q^\lambda \quad \text{and} \tag{41}$$

$$L_0^{2,i} \leftarrow x^\lambda(q^\lambda), \tag{42}$$

where

$$L_0^{1,\lambda} = q^\lambda \quad \text{and} \tag{43}$$

$$L_0^{2,\lambda} = x^\lambda(q^\lambda) \tag{44}$$

are the Lévy flight positions of the neighbor individual

$$F_\lambda^j = \left(L^{1,\lambda}, L^{2,\lambda}\right) \in C_j \tag{45}$$

with currently best test case quality evaluation $\tilde{E}_\lambda^j$ among neighbors in C_j, $(j = 1, \ldots, k)$.

7 Implementation

To show the feasibility of our approach, we implemented a prototype for the proposed self-adaptive fuzzing algorithm (as depicted in Fig. 1). Our implementation is based on Intel's dynamic instrumentation tool Pin [26] to trace the reached basic blocks of a generated test case. In order to calculate the number $E(x_0, \mathcal{I}')$ of newly discovered basic blocks executed by a test case x_0 as defined in Eq. (14), we switch off *Address Space Layout Randomization* (ASLR) during testing. For developing exploits based on a malicious input x_0 ASLR should naturally be enabled again.

Initially, we simulated the Lévy flights in the statistical computing language R [27] but then changed to a custom sampling method purely written in Python. We construct Lévy flights by summing up independent and identically distributed random variables as indicated in Eq. (5). Each addend is distributed according to a power law as defined in Eq. (12). We realize this by applying the *inverse transform sampling* method, also referred to as *Smirnov transform*. The Python script further performs evaluation of the current path exploration performance by direct comparison of executed basic block addresses received from dynamic instrumentation.

We implemented fuzzing swarms by parallel execution of multiple individual fuzzers which are clustered and relocated according to the algorithm described in Section 6. For clustering, we apply the Lloyd k-means algorithm.

In our implementation, we omit the first step $Last(X_{\text{all}})$ within the loop and instead always keep the last reached positions of the processes $(L_t^i)_{t\in\mathbb{N}}$ $(i = 1, 2)$ between simulations. This is due to the construction of new test cases in Eqs. (20)-(23) so that the last test case within X_{all} is simply the most recently generated $x_{k_{\text{gen}}}$ which will be used as starting position within the subsequent loop iteration. Therefore it suffices to stop the Lévy flights after k_{gen} steps, save their current position, and proceed with adapted diffusivity parameters in the subsequent invocation of the *Gen()* function.

8 Discussion

In this section, we discuss properties, possible modifications, and expansions of our proposed fuzzing algorithm.

As demonstrated in Section 5, our algorithm is self-adaptive in the sense that it automatically focuses its search when reaching high quality regions in input space and widens exploration in case of low-quality input regions. One possible pitfall of such a self-adaptive property is the occurrence of attracting regions: if the Lévy flights $(L_t^i)_{t\in\mathbb{N}}$ $(i = 1, 2)$ enter regions of high quality and get the response from the quality evaluation step to focus their search (by decreasing their diffusivity), an improper quality rating mechanism might cause the Lévy flights to stay there forever. However, our evaluation method (as defined in Section 4) avoids this by favoring test cases that lead the target binary to execute undiscovered basic blocks and in turn devaluates inputs that lead to already known execution paths. Therefore, if the test case generation module gets feedback that it is currently exploring a region of high quality it focuses its search as long as new execution paths are detected. As soon as exploration of new execution paths stagnates, the feedback from the evaluation module switches to a low rating. Such a negative feedback again increases diffusivity according to Eqs. (26) and (27), which again causes the processes $(L_t^i)_{t\in\mathbb{N}}$ $(i = 1, 2)$ to diffuse into other regions of the input space.

Our swarm algorithm for multiple individual fuzzers in Section 6 is designed to be flexible, robust, decentralized, and self-organized. The fuzzing swarm is *flexible* in the sense that it adapts to perturbations caused by the nature of Lévy flights and the targeted binary: if an individual fuzzer enters super-diffusion and performs frequent large steps, it simply gets assigned to a new neighborhood in the next clustering step. The swarm is *robust* in the sense that it can deal with loss easily: if an individual fuzzer gets stuck because the target crashed, the swarm algorithm simply omits this individual in the next clustering step. While clustering and relocation is realized by a central component, all individual fuzzers are independent stochastic processes F^i $(i = 1, \ldots, d)$ which evolve *decentralized*. Finally, paths to bugs in the target emerge *self-organized* during the fuzzing process and are not predefined in any way.

One main modification of our algorithm (for individual fuzzers) would be interchanging the aim of maximizing code coverage with an adequate objective. In Section 4, we defined a quality measure for generated test cases based on the number of new basic blocks we reach with those inputs. Although this is the most common strategy when searching for bugs in a target program of unknown structure, we could apply other objectives. For example, we could aim for triggering certain data flow relationships, executing preferred regions of code, or reach a predefined class of statements within the code. Our fuzzing algorithm is modular and flexible in that it allows to interchange the quality measure according to different testing objectives. More examples of such testing objectives are discussed in the field of test case prioritization (e.g., in [20, 21]).

9 Conclusions

Inspired by moving patterns of foraging animals, we introduce the first self-adaptive fuzzer based on Lévy flights. Just like search patterns in biology have evolved to optimal foraging strategies due to natural selection, so have evolved mathematical models to describe those patterns. Lévy flights are emerging as successful models

for describing optimal search behavior, which leads us to their application of hunting bugs in binary executables. By defining corresponding stochastic processes within the input space of the program under test, we achieve an effective new method for test case generation. Further, we define an algorithm that dynamically controls diffusivity of the defined Lévy flights depending on actual quality of generated test cases. To achieve this, we construct a measure of quality for new test cases that takes already explored execution paths into account. During fuzzing, the quality of actually generated test cases is constantly forwarded to the test case generating Lévy flights. High-quality test case generation with respect to path coverage causes the Lévy flight to enter sub-diffusion and focus its search on nearby inputs, whereas a low-quality rating results in super-diffusion and expanding search behavior. This feedback loop yields a fully self-adaptive fuzzer. Inspired by the collective behavior of certain animal colonies, we aggregate multiple individual fuzzers to a fuzzing swarm which is guided by simple rules to reveal flexible, robust, decentralized, and self-organized behavior. Our proposed algorithm is modular in the sense that it allows integration of other fuzzing goals beyond code coverage, which is subject to future work.

Competing interests
The author declares that he has no competing interests.

Received: 7 September 2016 Accepted: 9 November 2016
Published online: 21 November 2016

References

1. C Cadar, K Sen, Symbolic execution for software testing: three decades later. Commun. ACM. **56**(2), 82–90 (2013)
2. CS Păsăreanu, W Visser, A survey of new trends in symbolic execution for software testing and analysis. Int J Soft Tools Technol. Transfer. **11**(4), 339–353 (2009)
3. K Sen, D Marinov, G Agha, in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering. ESEC/FSE-13*. CUTE: a concolic unit testing engine for C (ACM, New York, 2005), pp. 263–272. doi:10.1145/1081706.1081750
4. P Godefroid, N Klarlund, K Sen, DART: Directed automated random testing. SIGPLAN Not. **40**(6), 213–223 (2005). doi:10.1145/1064978.1065036
5. P Godefroid, MY Levin, D Molnar, SAGE: whitebox fuzzing for security testing. Commun. ACM. **55**(3), 40 (2012). doi:10.1145/2093548.2093564
6. A Takanen, J DeMott, C Miller, *Fuzzing for Software Security Testing and Quality Assurance. 1st edn.* (Artech House, Inc., Norwood, 2008)
7. M Sutton, A Greene, P Amini, *Fuzzing: Brute Force Vulnerability Discovery. 1st edn.* (Addison-Wesley Professional, Boston, 2007)
8. F Lenz, TC Ings, L Chittka, AV Chechkin, R Klages, Spatiotemporal dynamics of bumblebees foraging under predation risk. Phys. Rev. Lett. **108**(9), 098103 (2012)
9. GM Viswanathan, Ecology: Fish in Lévy-flight foraging. Nature. **465**(7301), 1018–1019 (2010)
10. NE Humphries, N Queiroz, JR Dyer, NG Pade, MK Musyl, KM Schaefer, DW Fuller, JM Brunnschweiler, TK Doyle, JD Houghton, et al., Environmental context explains Lévy and Brownian movement patterns of marine predators. Nature. **465**(7301), 1066–1069 (2010)
11. D Austin, WD Bowen, JI McMillan, Intraspecific variation in movement patterns: modeling individual behaviour in a large marine predator. Oikos. **105**, 15–30 (2004). doi:10.1111/j.0030-1299.1999.12730.x
12. G Ramos-Fernández, JL Mateos, O Miramontes, G Cocho, H Larralde, B Ayala-Orozco, Lévy walk patterns in the foraging movements of spider monkeys (Ateles geoffroyi). Behav. Ecol. Sociobiol. **55**(3), 223–230 (2004)
13. GM Viswanathan, V Afanasyev, S Buldyrev, E Murphy, P Prince, HE Stanley, et al., Lévy flight search patterns of wandering albatrosses. Nature. **381**(6581), 413–415 (1996)
14. A Mårell, JP Ball, A Hofgaard, Foraging and movement paths of female reindeer: insights from fractal analysis, correlated random walks, and Lévy flights. Can. J. Zoology-Revue Canadienne De Zoologie. **80**, 854–865 (2002). doi:10.1139/z02-061
15. O Bénichou, C Loverdo, M Moreau, R Voituriez, Intermittent search strategies. Rev. Mod. Phys. **83**(1), 81 (2011)
16. TH Harris, EJ Banigan, DA Christian, C Konradt, EDT Wojno, K Norose, EH Wilson, B John, W Weninger, AD Luster, et al., Generalized Lévy walks and the role of chemokines in migration of effector CD8+ T cells. Nature. **486**(7404), 545–548 (2012)
17. GM Viswanathan, MG Da Luz, EP Raposo, HE Stanley, *The Physics of Foraging: an Introduction to Random Searches and Biological Encounters*. (Cambridge University Press, Cambridge, 2011)
18. E Bonabeau, M Dorigo, G Theraulaz, *Swarm Intelligence: From Natural to Artificial Systems*. (Oxford University Press, Inc., New York, 1999)
19. K Böttinger, in *2016 IEEE Security and Privacy Workshops (SPW)*. Hunting bugs with Lévy flight foraging (IEEE Computer Society, Los Alamitos, 2016), pp. 111–117. doi:10.1109/SPW.2016.9
20. D Leon, A Podgurski, in *Proceedings of the 14th International Symposium on Software Reliability Engineering*. A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases (IEEE Computer Society, Washington, DC, 2003), pp. 442–456. doi:10.1109/ISSRE.2003.1251065
21. G Rothermel, RH Untch, C Chu, MJ Harrold, in *Proceedings of the IEEE International Conference on Software Maintenance*. Test case prioritization: an empirical study (IEEE Computer Society, Washington, DC, 1999), pp. 179–188. doi:10.1109/ICSM.1999.792604
22. A Rebert, SK Cha, T Avgerinos, J Foote, D Warren, G Grieco, D Brumley, in *Proceedings of the 23rd USENIX Conference on Security Symposium*. Optimizing seed selection for fuzzing (USENIX Association, Berkeley, 2014), pp. 861–875
23. SK Cha, M Woo, D Brumley, in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*. Program-adaptive mutational fuzzing (IEEE Computer Society, Washington, DC, 2015), pp. 725–741. doi:10.1109/SP.2015.50
24. V Zaburdaev, S Denisov, J Klafter, Lévy walks. Rev. Mod. Phys. **87**, 483–530 (2015). doi:10.1103/RevModPhys.87.483
25. M Chupeau, O Bénichou, R Voituriez, Cover times of random searches. Nat. Phys. **11**, 844–847 (2015). Nature Publishing Group
26. C-K Luk, R Cohn, R Muth, H Patil, A Klauser, G Lowney, S Wallace, VJ Reddi, K Hazelwood, in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '05*. Pin: Building customized program analysis tools with dynamic instrumentation (ACM, New York, 2005), pp. 190–200. doi:10.1145/1065010.1065034
27. R Development Core Team, *R: A Language and Environment for Statistical Computing*. (R Foundation for Statistical Computing, Vienna, 2008)